A Generalized Program Verification Workflow Based on Loop Elimination and SA Form

Cláudio Belo Lourenço*, Maria João Frade† and Jorge Sousa Pinto†
*LRI, Université Paris-Sud & INRIA Saclay, France
†HASLab/INESC TEC & Universidade do Minho, Portugal

***Abstract*—This paper presents a minimal model of the functioning of program verification and property checking tools based on (i) the encoding of loops as non-iterating programs, either *conservatively*, making use of invariants and *assume/assert* commands, or in a *bounded* way; and (ii) the use of an intermediate *single-assignment* (SA) form. The model captures the basic workflow of tools like Boogie, Why3, or CBMC, building on a clear distinction between operational and axiomatic semantics. This allows us to consider separately the soundness of program annotation, loop encoding, translation into SA form, and verification condition (VC) generation, as well as appropriate notions of completeness for each of these processes.**

To the best of our knowledge, this is the first formalization of a bounded model checking of software technique, including soundness and completeness proofs using Hoare logic; we also give the first completeness proof of a deductive verification technique based on a conservative encoding of invariant-annotated loops with *assume/assert* in SA form, as well as the first soundness proof based on a program logic.

I. Introduction

The goal of this paper is to formalize and establish properties of key aspects of modern program checking tools, in particular deductive program verifiers and bounded model checkers of software. For reasons that are discussed below, it turns out that many tools eliminate loops and convert the resulting code into an intermediate single-assignment form.

1) Single-assignment Form for Checking Programs: Single Assignment (SA) programs impose the restriction that each variable cannot be assigned after it has been used (read or assigned). They have been introduced as intermediate forms in compilation pipelines, in both static [8] and dynamic [27] variants. More recently they have been used as intermediate forms in the context of program verification and bug finding tools, for a number of reasons. Programs in SA form are easy to encode in logic, since assignments can be written directly as equalities: the instruction $x := x + 1$ cannot be encoded as the formula $x = x + 1$, but its single-assignment counterpart $x_2 := x_1 + 1$ can be encoded as $x_2 = x_1 + 1$. This encoding is commonly employed in tools such as CBMC [6], that check for assertion violations by solving satisfiability problems. Furthermore, the resulting logical encoding is *compact*. The number of execution paths of a program may grow exponentially with its size, and encodings that directly follow the control flow will generate formulas of exponential size – this is the case in some symbolic execution tools [1], and tools based on predicate transformers such as *weakest precondition*. For a certain class of programs this exponential explosion can be avoided [14], by first translating them into an SA form.

There are additional reasons for the use of SA form. An operator that is commonly used in the source language of program verifiers is known as *havoc*; it has the effect of modifying the value of a variable in a non-deterministic way, and is an essential part of conservative iteration-free encodings of loops. Since in the single-assignment intermediate form each variable x is represented by a family of variables $\{x_i\}$, whenever the value of x needs to be *havoced* it suffices to start using a fresh variable x_k in the SA form.

Finally, in an SA setting the need for *auxiliary variables* may be greatly reduced (or even eliminated). Note that the initial and all intermediate values of every variable of the original program are available throughout the entire execution of the SA form (since variables are only assigned once), which eliminates the need to use auxiliary variables to record these initial or intermediate values, or to consider labels in the program semantics [16]. See for instance [9], where the authors show examples of verified programs that require the use of auxiliary variables, and describe the advantages of eliminating such variables.

2) Checking Iterating Code: A major issue in program verification is the encoding of iterating programs. Iterating constructs are not present in SA form, and must be eliminated when programs are converted to this intermediate form. Two different families of tools, and major approaches to reasoning about loops, are:

- *Deductive program verifiers* (DV), based on the axiomatic semantics of programs, which rely on the use of *loop invariants* (typically provided by users).
- *Bounded model checkers of software* (BMC), which eliminate loops by unrolling them a given number of times.

In BMC, loops are removed by the very nature of the technique: the branching programs resulting from loop unfolding can be readily converted to SA form. But in the deductive approach an additional step is required, to convert loops annotated with invariants into iteration-free programs that can be expressed in SA form. The most widely used technique, employed by deductive tools like Boogie [2] and Why3 [12], is to encode loops by means of *assume* and *assert* commands, using the *havoc* operator (described above) in order to isolate different parts of the encoding. However, if loop elimination is performed simultaneously with conversion to SA form, the havoc operator is not required. As an example, let Fact be the

```
f := 1 ; i := 1 ;
while i ≤ n do {f = (i − 1)! ∧ i ≤ n + 1}
  f := f ∗ i ;
  i := i + 1
od
```

(a) Factorial program

```
f := 1 ; i := 1 ;
if i ≤ n then
  f := f ∗ i ; i := i + 1 ;
  if i ≤ n then
    f := f ∗ i ; i := i + 1 ;
    assert ¬(i ≤ n) fi fi
```

(b) BMC encoding of Factorial

```
f := 1 ; i := 1 ;
assert f = (i − 1)! ∧ i ≤ n + 1 ;
f := havoc ; i := havoc ;
assume f = (i − 1)! ∧ i ≤ n + 1 ;
if i ≤ n then
  f := f ∗ i ; i := i + 1 ;
  assert f = (i − 1)! ∧ i ≤ n + 1 ;
  assume ⊥ fi
```

(c) *Havoc* encoding of Factorial

```
f1 := 1 ; i1 := 1 ;
assert f1 = (i1 − 1)! ∧ i1 ≤ n0 + 1 ;
assume f2 = (i2 − 1)! ∧ i2 ≤ n0 + 1 ;
if i2 ≤ n0 then
  f3 := f2 ∗ i2 ; i3 := i2 + 1 ;
  assert f3 = (i3 − 1)! ∧ i3 ≤ n0 + 1 ;
  assume ⊥
else f3 := f2 ; i3 := i2 fi
```

(d) SA encoding of Factorial (obtained after Havoc encoding)

Fig. 1: An example

factorial program of Figure 1a, annotated with a loop invariant. It can be encoded without iteration as shown in Figure 1c, or in SA form, Figure 1d. Figure 1b shows a bounded encoding (straightforward to convert to SA form).

3) Contributions: We formalize the deductive and bounded approaches to program verification by means of a workflow consisting of the following steps (see Figure 2). The same SA translation function handles annotated loops (in the DV workflow) and iteration-free programs (in the BMC workflow).

(i) *Loop unfolding or annotation*: loops are either annotated with loop invariants (the DV approach), or else they are unfolded a given number of times (the BMC approach).
(ii) *SA translation*: the program resulting from step (i) is translated into SA form. If annotated loops are present they will in this step be eliminated, encoded by means of *assume* and *assert* statements.
(iii) *Verification Condition Generation*: a set of logical formulas is generated from the SA form, such that if all formulas are valid then the program is correct.

The notion of correctness that will be considered is that the program is in accordance with a specification given by a set of *assume* and *assert* commands included in it, together with a precondition and a pair of postconditions (corresponding to normal and exceptional termination). The properties that are expected of a tool for checking correctness are *soundness* (every property violation should be identified) and *completeness* (no false positives should be found – a property that can only be established in a relative sense). In bounded verification it is not possible to have both properties simultaneously – the user must choose to unfold loops in a way that allows for either soundness or completeness.

Basing our formalization on the axiomatic semantics of the programming language allows us to express these properties rigorously. As far as we know, this is the first formalization of a bounded model checking of software technique, including soundness and completeness proofs. We also give, for a deductive verification technique based on the use of SA form and loop encoding using *assume/assert* (as used in practice by major verification tools), the first completeness proof as well as the first soundness proof based on program logic.

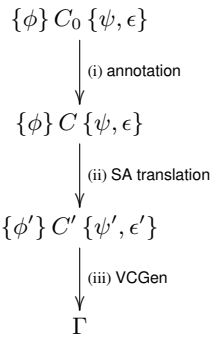
The paper is organized as follows: the next section gives necessary background and introduces the source language that will be used throughout the paper. In Section III we define the intermediate SA language, as well as a Hoare logic for it. Section IV characterizes the two components of the DV workflow, (a translation into SA form and a verification condition generator – VCGen – for SA programs) and proves its soundness and completeness. Section V is devoted to the BMC workflow, building on the previously defined notion of SA translation, and adding loop unwinding functions. Soundness or completeness of the workflow are established, depending on how the unwinding is carried out. Finally, Section VI discusses related work and Section VII concludes the paper.

II. Iterating Programs with Exceptions and Assume/Assert Commands

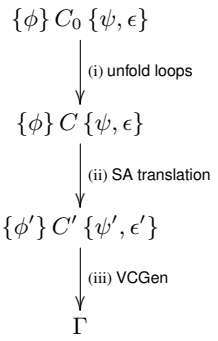
1) Syntax: Throughout this paper a While language with exceptions and also assume and assert commands (as normally found in the guarded commands language introduced by Dijkstra [11]) will be considered as source language:

$$\begin{aligned}\mathbf{Comm} \ni C ::= {} & \mathbf{skip} \mid \mathbf{throw} \mid x := e \mid \mathbf{assume}\,\theta \mid \mathbf{assert}\,\theta \\ & \mid C\,;\,C \mid \mathbf{try}\,C\,\mathbf{catch}\,C\,\mathbf{hc} \\ & \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}\end{aligned}$$

The programs are constructed over a set of variables $x \in \mathbf{Var}$ and a language of program expressions $e \in \mathbf{Exp}$ and Boolean expressions $b \in \mathbf{Exp_{bool}}$ that will not be fixed here (a standard instantiation is for $\mathbf{Exp}$ to be a language of integer expressions, with $\mathbf{Exp_{bool}}$ constructed from comparison operators over $\mathbf{Exp}$ and Boolean operators). In addition, we require program assertions $\theta \in \mathbf{Assert}$, obtained as a first-order expansion of $\mathbf{Exp_{bool}}$, to express properties about states. We use the following notation: the sets of *variables occurring* and *assigned* in the program C are written $\mathsf{Vars}(C)$ and $\mathsf{Asgn}(C)$ respectively; $\mathsf{Vars}(e)$ and $\mathsf{Vars}(b)$ are, respectively, the sets of



(a) DV workflow
Program C contains loops annotated with invariants



(b) BMC workflow
Program C does not contain loops

Fig. 2: Verification Workflows. C_0 is an iterating program, ϕ its precondition, and ψ, ϵ are its normal and exceptional postconditions; C' is a single-assignment program; Γ is a set of first-order formulas (verification conditions)

1) $\langle \mathbf{skip}, s\rangle \Rightarrow \mathbf{n}(s)$.
2) $\langle \mathbf{throw}, s\rangle \Rightarrow \mathbf{e}(s)$.
3) $\langle x := e, s\rangle \Rightarrow \mathbf{n}(s[x \mapsto [\![e]\!](s)])$.
4) If $s \models \theta$, then $\langle \mathbf{assume}\,\theta, s\rangle \Rightarrow \mathbf{n}(s)$.
5) If $s \models \theta$, then $\langle \mathbf{assert}\,\theta, s\rangle \Rightarrow \mathbf{n}(s)$.
6) If $s \not\models \theta$, then $\langle \mathbf{assert}\,\theta, s\rangle \Rightarrow \bullet$.
7) If $\langle C_1, s\rangle \Rightarrow \bullet$, then $\langle C_1\,;\, C_2, s\rangle \Rightarrow \bullet$.
8) If $\langle C_1, s\rangle \Rightarrow \mathbf{e}(s')$, then $\langle C_1\,;\, C_2, s\rangle \Rightarrow \mathbf{e}(s')$.
9) If $\langle C_1, s\rangle \Rightarrow \mathbf{n}(s')$, then $\langle C_1\,;\, C_2, s\rangle \Rightarrow \langle C_2, s'\rangle$.
10) If $\langle C_1, s\rangle \Rightarrow \langle C_1', s'\rangle$, then $\langle C_1\,;\, C_2, s\rangle \Rightarrow \langle C_1'\,;\, C_2, s'\rangle$.
11) If $\langle C_1, s\rangle \Rightarrow \bullet$, then $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \bullet$.
12) If $\langle C_1, s\rangle \Rightarrow \mathbf{e}(s')$, then $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \langle C_2, s'\rangle$.
13) If $\langle C_1, s\rangle \Rightarrow \mathbf{n}(s')$, then $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \mathbf{n}(s')$.
14) If $\langle C_1, s\rangle \Rightarrow \langle C_1', s'\rangle$, then $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \langle \mathbf{try}\, C_1'\, \mathbf{catch}\, C_2\, \mathbf{hc}, s'\rangle$.
15) If $s \models b$, then $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \Rightarrow \langle C_1, s\rangle$.
16) If $s \not\models b$, then $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \Rightarrow \langle C_2, s\rangle$.
17) $\langle \mathbf{while}\ b\ \mathbf{do}\ C_1\ \mathbf{od}, s\rangle \Rightarrow \langle \mathbf{if}\ b\ \mathbf{then}\ C_1\,;\, \mathbf{while}\ b\ \mathbf{do}\ C_1\ \mathbf{od}\ \mathbf{else}\ \mathbf{skip}\ \mathbf{fi}, s\rangle$.

Fig. 3: Operational (small-step) semantics for $\mathbf{Comm}$

variables occurring in expressions e and b; and $\mathsf{FV}(\theta)$ denotes the set of free variables in the assertion θ (all are defined as expected, noting that $\mathsf{Vars}(C)$ includes the free variables of the assertions contained in C).

Specifications are tuples (ϕ, ψ, ϵ), with $\phi, \psi, \epsilon \in \mathbf{Assert}$: ϕ is the *precondition* (assumed to hold when the program is executed), whereas ψ and ϵ are the *postconditions* (required to hold when program execution stops normally or exceptionally, respectively). A *Hoare triple* [18], written as $\{\phi\}\, C\, \{\psi, \epsilon\}$, expresses the fact that the program C conforms to the specification (ϕ, ψ, ϵ). It should be noted that assume and assert commands are not mandatory at source level (although they will allow to express additional specification constraints). On the other hand, in the *intermediate-language* they play an essential role in the encoding (both bounded and unbounded) of iterating constructs.

2) Semantics: For the vocabulary describing the concrete syntax of program expressions, we will consider an *interpretation structure* $\mathcal{M} = (D, I)$. This structure provides an interpretation domain D as well as a concrete interpretation of constants and operators, given by I. The interpretation of expressions depends on a *state*, which is a function that maps each variable into a value. We will write $\Sigma = \mathbf{Var} \to D$ for the set of states. For $s \in \Sigma$, $s[x \mapsto a]$ will denote the state that maps x to a and any other variable y to $s(y)$.

The interpretation of $e \in \mathbf{Exp}$ (resp. $b \in \mathbf{Exp_{bool}}$) in $\mathcal{M}$ will be given by a function $[\![e]\!]_{\mathcal{M}} : \Sigma \to D$ (resp. $[\![b]\!]_{\mathcal{M}} : \Sigma \to \{\bot, \top\}$), thus every expression has a value at every state, and expression evaluation does not modify the state. For assertions we take the usual interpretation of first-order formulas, noting that since assertions build on the language of program expressions their interpretation also depends on $\mathcal{M}$. The interpretation of the assertion $\phi \in \mathbf{Assert}$, using states from Σ as *variable assignments*, is then given by $[\![\phi]\!]_{\mathcal{M}} : \Sigma \to \{\bot, \top\}$, and we will write $s \models \phi$ as shorthand for $[\![\phi]\!]_{\mathcal{M}}(s) = \top$. For the sake of readability, we will omit the $\mathcal{M}$ subscript, leaving the interpretation structure implicit. When $s \models \phi$ for all $s \in \Sigma$, ϕ is said to be *valid*, written $\models \phi$. For a set of assertions Γ, we write $\models \Gamma$ if $\models \phi$ for every $\phi \in \Gamma$.

Let us now focus on the interpretation of programs. Semantically, the command $\mathbf{assume}\,\theta$ is seen as blocking whenever executed in a state in which θ is false, and $\mathbf{assert}\,\theta$ is seen as producing an error when θ is false. Moreover, as we have exceptions, we have to distinguish states representing normal termination from states representing exceptional termination. We will let $\Sigma_\bullet \ni \sigma ::= \mathbf{n}(s) \mid \mathbf{e}(s) \mid \bullet$ be the set of possible final states: with $s \in \Sigma$, $\mathbf{n}(s)$ is a *normal* termination state, $\mathbf{e}(s)$ is an *exceptional* termination state (reached by executing $\mathbf{throw}$), and $\bullet$ is the *error* state (reached by executing $\mathbf{assert}\,\theta$ when θ does not hold). We consider the structural operational semantics given by the deterministic transition relation $\Rightarrow \subseteq \mathbf{Comm} \times \Sigma \times (\Sigma_\bullet + \mathbf{Comm} \times \Sigma)$ defined in Figure 3 (which again depends on an implicit interpretation of program expressions). A configuration can evolve into a final state, progress into an intermediate configuration leaving part of the program to be evaluated, or simply get stuck. We will write $\gamma \Rightarrow^n \gamma'$ to indicate that there are n steps in the execution from configuration γ to γ', and write $\langle C, s\rangle \not\Rightarrow$ to denote that the program C cannot evolve from state s, i.e., that $\langle C, s\rangle$ is a stuck configuration ($\langle \mathbf{assume}\,\theta, s\rangle \not\Rightarrow$ when $s \not\models \theta$). As usual, $\gamma \Rightarrow^* \gamma'$ (resp. $\gamma \Rightarrow^+ \gamma'$) denotes that there are zero or more (resp. one or more) steps in the execution from γ to γ'.

Turning now to Hoare triples, we must adapt the notion of validity to programs containing exceptions and assume/assert commands. These are programs that already contain their own built-in specification, conferred by the assume and assert statements. Our interpretation of Hoare triples must handle this 'internal' specification in addition to the 'external' specifica-

tion given by the precondition ϕ and postconditions ψ, ϵ. With these aspects in mind, expressing the validity of a Hoare triple requires stating that executions that terminate (i.e., do not get stuck) do not enter the $\bullet$ state (because of a failed assert), and also satisfy the normal or exceptional postcondition, depending on the termination.

Definition 1 (Validity of Hoare triples): A Hoare triple $\{\phi\}\, C\, \{\psi, \epsilon\}$ is said to be *valid*, denoted $\models \{\phi\}\, C\, \{\psi, \epsilon\}$, whenever for every $s \in \Sigma$ and $\sigma \in \Sigma_\bullet$, if $s \models \phi$ and $\langle C, s\rangle \Rightarrow^* \sigma$ then:

1) $\sigma \neq \bullet$;
2) if $\sigma = \mathbf{n}(s')$ for some $s' \in \Sigma$, then $s' \models \psi$;
3) if $\sigma = \mathbf{e}(s')$ for some $s' \in \Sigma$, then $s' \models \epsilon$.

Note that this is a *partial notion of correctness*, since it does not require termination. We will focus on this notion for the sake of simplicity.

3) Hoare Calculus: The standard Hoare logic [18] inference system can be extended to Hoare triples with exceptions, assumes and asserts. This system, which we call H, is shown in Figure 4 (top) and contains the rule (conseq) that is guarded by first-order side conditions, whose validity must be checked when constructing derivations. We will consider that reasoning in this system takes place in the context of the *complete theory* $\mathrm{Th}(\mathcal{M})$ of the implicit structure $\mathcal{M}$, therefore when constructing derivations in H one simply checks, when applying the (conseq) rule, whether the side conditions are elements of $\mathrm{Th}(\mathcal{M})$. As a result, we will write $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$ as a shorthand for $\mathrm{Th}(\mathcal{M}) \vdash_{\mathsf{H}} \{\phi\}\, \mathrm{C}\, \{\psi, \epsilon\}$, denoting the fact that the triple is derivable in this system with $\mathrm{Th}(\mathcal{M})$.

System H admits multiple derivations for the same Hoare triple, and does not force a particular strategy for constructing them. However, the (assign) rule is based on a weakest precondition calculation, and as such, derivations based on backward propagation are in a sense more natural in this system. The (assume) and (assert) rules follow the assignment rule in this respect: they propagate the normal postcondition ψ backward, according to the definition of the weakest precondition predicate transformer for the guarded commands language [11].

We will now show that the fundamental properties of Hoare logic for the While language (without exceptions and assume/assert) extend to system H. The system is sound wrt. the semantics of Hoare triples; it is also complete in the sense of Cook [7], i.e. as long as the assertion language is sufficiently *expressive*. One way to ensure the expressiveness of our language is to force the existence of both a normal and an exceptional strongest postcondition for every command and assertion. Let $C \in \mathbf{Comm}$ and $\phi \in \mathbf{Assert}$, and denote respectively by $\mathrm{postN}(\phi, \mathrm{C})$ and $\mathrm{postE}(\phi, \mathrm{C})$ the sets of states $\{s' \in \Sigma \mid \langle C, s\rangle \Rightarrow^* \mathbf{n}(s') \text{ for some } s \in \Sigma \text{ such that } s \models \phi\}$ and $\{s' \in \Sigma \mid \langle C, s\rangle \Rightarrow^* \mathbf{e}(s') \text{ for some } s \in \Sigma \text{ such that } s \models \phi\}$. The assertion language $\mathbf{Assert}$ is said to be *expressive* with respect to the command language $\mathbf{Comm}$ and interpretation structure $\mathcal{M}$, if for every $\phi \in \mathbf{Assert}$ and $C \in \mathbf{Comm}$ there exist $\psi, \epsilon \in \mathbf{Assert}$ such that for any $s \in \Sigma$, (i) $s \models \psi$ iff $s \in \mathrm{postN}(\phi, \mathrm{C})$ and (ii) $s \models \epsilon$ iff $s \in \mathrm{postE}(\phi, \mathrm{C})$. The properties of system H can now be expressed as follows, and proved respectively by induction on the structure of $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$ and on the structure of C. See [25] for full proofs in this setting with exceptions.

Proposition 1 (Soundness of system H*):* Let $C \in \mathbf{Comm}$ and $\phi, \psi, \epsilon \in \mathbf{Assert}$. If $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$, then $\models \{\phi\}\, C\, \{\psi, \epsilon\}$.

Proposition 2 (Completeness of system H *in the sense of Cook):* Let $C \in \mathbf{Comm}$, $\phi, \psi, \epsilon \in \mathbf{Assert}$ such that $\mathbf{Assert}$ is expressive wrt. $\mathbf{Comm}$ and the implicit interpretation structure. If $\models \{\phi\}\, C\, \{\psi, \epsilon\}$, then $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$.

4) A Goal-directed Inference System Guided by Loop-invariant Annotations: We will also require a syntactic class $\mathbf{AComm}$ of *annotated programs*, which differs from $\mathbf{Comm}$ only in the case of the loop construct. The new construct is annotated with a loop invariant θ and has the form $\mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}$. Annotations do not affect the operational semantics. Note that for $C \in \mathbf{AComm}$, $\mathsf{Vars}(C)$ now includes the free variables of the annotations contained in C. In what follows we will use the auxiliary function $\lfloor\cdot\rfloor : \mathbf{AComm} \to \mathbf{Comm}$ that erases all annotations from a program (defined in the obvious way).

In Figure 4 (bottom) we present system Hg, an inference system for Hoare triples $\{\phi\}\, C\, \{\psi, \epsilon\}$ where $C \in \mathbf{AComm}$. This system is intended for the *mechanical construction* of derivations, which is essential to reason about the generation of verification conditions, covered by our workflows. Loop invariants are not invented at this point, but rather taken from the annotations, and there is no ambiguity in the choice of rule to apply, since a consequence rule is not present. The different possible derivations for a given triple in Hg differ only in the intermediate assertions that are used in the (seq) and (try-catch) rules.

It is easy to see that the (conseq) rule is admissible in system Hg, and that systems H and Hg are in fact equivalent.

Proposition 3 (Soundness of Hg*):* If $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$, then $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor C \rfloor\, \{\psi, \epsilon\}$.

Proof. By induction on the derivation of $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$. □

Lemma 1: If $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$, $\models \phi' \to \phi$, $\models \psi \to \psi'$, and $\models \epsilon \to \epsilon'$, then $\vdash_{\mathsf{Hg}} \{\phi'\}\, C\, \{\psi', \epsilon'\}$.

Proof. By induction on the derivation of $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$. □

Proposition 4 (Completeness of Hg*):* If $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$, then there exists some $C' \in \mathbf{AComm}$ such that $\lfloor C' \rfloor = C$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, C'\, \{\psi, \epsilon\}$.

Proof. If $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$, then there exists at least one derivation $\mathcal{D}$ with conclusion $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$. Let C' be the C program with the while instructions annotated with the same invariants used in the derivation $\mathcal{D}$. The proof of $\vdash_{\mathsf{Hg}} \{\phi\}\, C'\, \{\psi, \epsilon\}$ follows by routine induction on $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$, using Lemma 1 for the case where the last rule applied is (conseq). □

III. Single-assignment Programs

In this section we define the notions of SA program and SA Hoare triple. An inference system for these triples is introduced, and its soundness and completeness are established.

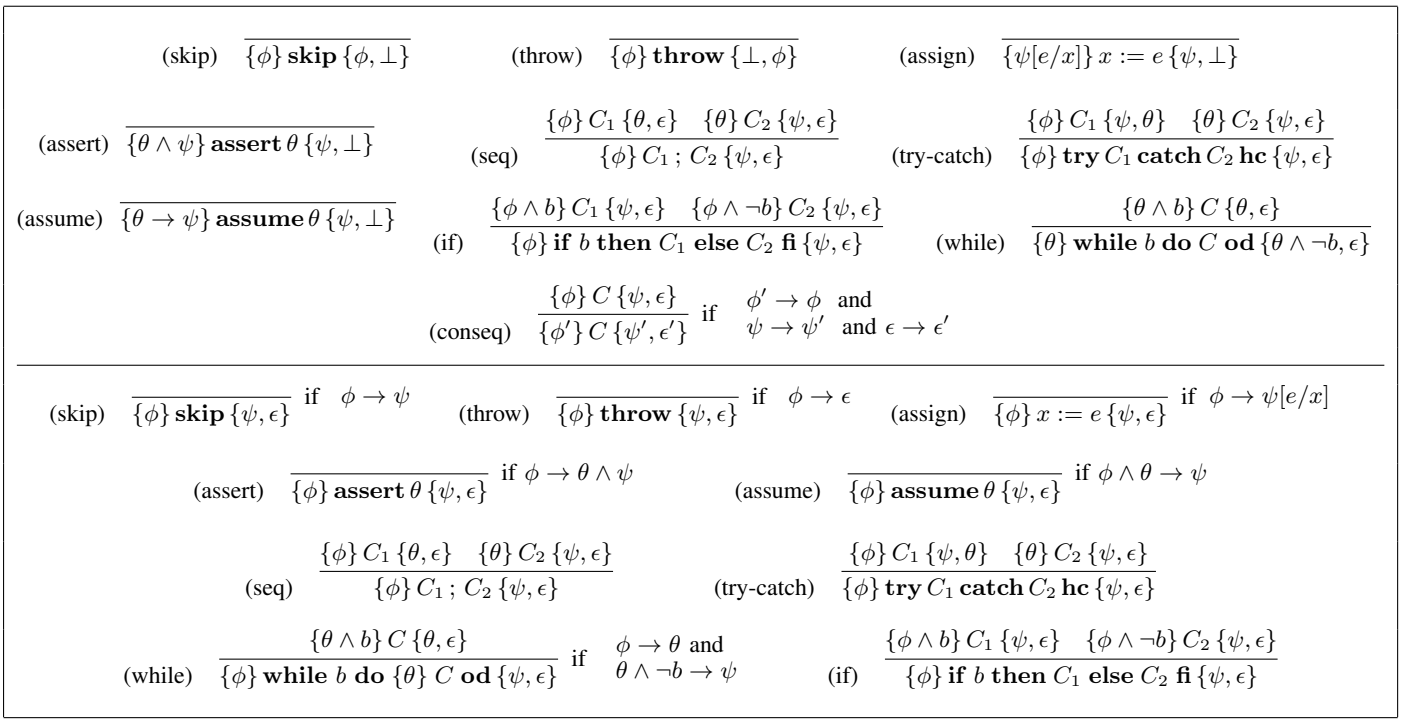


Fig. 4: Systems H (top) and Hg (bottom)

Definition 2 (SA program): The set $\mathbf{Comm}^{\text{SA}} \subset \mathbf{Comm}$ of *single-assignment programs* is defined inductively as follows:

- $\mathbf{skip}, \mathbf{assert}\,\theta, \mathbf{assume}\,\theta, \mathbf{throw} \in \mathbf{Comm}^{\text{SA}}$;
- $x := e \in \mathbf{Comm}^{\text{SA}}$ if $x \notin \mathsf{Vars}(e)$;
- $C_1\,;\,C_2 \in \mathbf{Comm}^{\text{SA}}$ if $C_1, C_2 \in \mathbf{Comm}^{\text{SA}}$, and $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$;
- $\mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc} \in \mathbf{Comm}^{\text{SA}}$ if $C_1, C_2 \in \mathbf{Comm}^{\text{SA}}$, and $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$;
- $\mathbf{if}\,b\,\mathbf{then}\,C_1\,\mathbf{else}\,C_2\,\mathbf{fi} \in \mathbf{Comm}^{\text{SA}}$ if $C_1, C_1 \in \mathbf{Comm}^{\text{SA}}$, and $\mathsf{Vars}(b) \cap (\mathsf{Asgn}(C_1) \cup \mathsf{Asgn}(C_2)) = \emptyset$.

This definition guarantees that variables are assigned at most once in any execution, and never after they have been read. We will extend this notion to Hoare triples. Let us write $\phi\#C$ to denote $\mathsf{Asgn}(C) \cap \mathsf{FV}(\phi) = \emptyset$. A triple $\{\phi\}\,C\,\{\psi,\epsilon\}$ with $C \in \mathbf{Comm}^{\text{SA}}$ is said to be *single-assignment* if $\phi\#C$, i.e. the program is not allowed to assign variables occurring free in the precondition (note that this does not restrict the power of specifications, since these variables would be assigned without their initial values ever being read).

Figure 5 contains the rules of the goal-directed system Hsa, which is based on *forward propagation* of assertions encoding executions of the program. It derives triples of the form $\{\phi\}\,C\,\{\phi\wedge\psi, \phi\wedge\epsilon\}$, where the precondition ϕ encodes logically a set of incoming executions and $\phi\wedge\psi$, $\phi\wedge\epsilon$ are respectively the normal and exceptional strongest postconditions of C with respect to ϕ, where the formulas ψ and ϵ encode all normal and exceptional executions of C. This system is sound wrt. system H. We start with two lemmas concerning derivability in Hsa.

confirmar se é usado

Lemma 2: If $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\phi\wedge\psi, \phi\wedge\epsilon\}$ and $\models \phi'\to\phi$, then $\vdash_{\mathsf{Hsa}} \{\phi'\}\,C\,\{\phi'\wedge\psi, \phi'\wedge\epsilon\}$.

Proof. By induction on $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\phi\wedge\psi, \phi\wedge\epsilon\}$. □

Lemma 3: Let $C \in \mathbf{Comm}^{\text{SA}}$ and $\phi, \psi, \psi', \epsilon, \epsilon' \in \mathbf{Assert}$ such that $\phi\#C$, and $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi,\epsilon\}$. Then:

1) $\mathsf{FV}(\psi) \cup \mathsf{FV}(\epsilon) \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$.
2) If $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi',\epsilon'\}$, then $\psi' = \psi$ and $\epsilon' = \epsilon$.

Proof. 1 follows by induction on $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi,\epsilon\}$ and 2 by induction on C. □

Proposition 5 (Soundness of Hsa*):* Let $C \in \mathbf{Comm}^{\text{SA}}$ and $\phi, \psi, \epsilon \in \mathbf{Assert}$ such that $\phi\#C$. If $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi,\epsilon\}$, then $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi,\epsilon\}$.

Proof. By induction on $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi,\epsilon\}$, using Lemma 3. □

The completeness of Hsa on the other hand will be established with respect to the goal-directed system Hg. Observe that the Hsa system is not capable of deriving every valid triple, and the completeness result takes this into account.

Proposition 6 (Completeness of Hsa*):* Let $C \in \mathbf{Comm}^{\text{SA}}$ and $\phi, \psi, \epsilon \in \mathbf{Assert}$ such that $\phi\#C$ and $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi,\epsilon\}$. Then $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\phi\wedge\psi', \phi\wedge\epsilon'\}$ for some $\psi', \epsilon' \in \mathbf{Assert}$ such that $\models \phi\wedge\psi'\to\psi$ and $\models \phi\wedge\epsilon'\to\epsilon$.

Proof. By induction on C, using Lemmas 1 and 3. □

This inference system will be used in the next sections as a reference for the definition of verification condition generators.

IV. Single-assignment Deductive Verification

We will now establish the soundness and relative completeness of a workflow for the deductive verification of While programs. This consists in, after first annotating loops with

(skip) $\dfrac{}{\{\phi\}\,\mathbf{skip}\,\{\phi\wedge\top,\phi\wedge\bot\}}$ (throw) $\dfrac{}{\{\phi\}\,\mathbf{throw}\,\{\phi\wedge\bot,\phi\wedge\top\}}$ (assign) $\dfrac{}{\{\phi\}\,x:=e\,\{\phi\wedge x=e,\phi\wedge\bot\}}$

(assert) $\dfrac{}{\{\phi\}\,\mathbf{assert}\,\theta\,\{\phi\wedge\theta,\phi\wedge\bot\}}$ if $\phi\to\theta$ (seq) $\dfrac{\{\phi\}\,C_1\,\{\phi\wedge\psi_1,\phi\wedge\epsilon_1\}\quad\{\phi\wedge\psi_1\}\,C_2\,\{\phi\wedge\psi_1\wedge\psi_2,\phi\wedge\psi_1\wedge\epsilon_2\}}{\{\phi\}\,C_1\,;\,C_2\,\{\phi\wedge(\psi_1\wedge\psi_2),\phi\wedge(\epsilon_1\vee(\psi_1\wedge\epsilon_2))\}}$

(assume) $\dfrac{}{\{\phi\}\,\mathbf{assume}\,\theta\,\{\phi\wedge\theta,\phi\wedge\bot\}}$ (try-catch) $\dfrac{\{\phi\}\,C_1\,\{\phi\wedge\psi_1,\phi\wedge\epsilon_1\}\quad\{\phi\wedge\epsilon_1\}\,C_2\,\{\phi\wedge\epsilon_1\wedge\psi_2,\phi\wedge\epsilon_1\wedge\epsilon_2\}}{\{\phi\}\,\mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}\,\{\phi\wedge(\psi_1\vee(\epsilon_1\wedge\psi_2)),\phi\wedge(\epsilon_1\wedge\epsilon_2)\}}$

(if) $\dfrac{\{\phi\wedge b\}\,C_1\,\{\phi\wedge b\wedge\psi_1,\phi\wedge b\wedge\epsilon_1\}\quad\{\phi\wedge\neg b\}\,C_2\,\{\phi\wedge\neg b\wedge\psi_2,\phi\wedge\neg b\wedge\epsilon_2\}}{\{\phi\}\,\mathbf{if}\;b\;\mathbf{then}\;C_1\;\mathbf{else}\;C_2\;\mathbf{fi}\,\{\phi\wedge((b\wedge\psi_1)\vee(\neg b\wedge\psi_2)),\phi\wedge((b\wedge\epsilon_1)\vee(\neg b\wedge\epsilon_2))\}}$

Fig. 5: System Hsa

invariants, translating programs to SA form and subsequently generating compact verification conditions.

The deductive workflow relies on two components. The first is a translation of annotated programs into SA form. The translation will in fact operate at the level of Hoare triples, rather than of isolated programs. Such a translation must comply with the syntactic restrictions of $\mathbf{Comm}^{\text{SA}}$, with additional requirements of a semantic nature. In particular, the translation must be sound (it will not translate invalid triples into valid ones). Moreover, SA programs will be annotated with loop invariants (obtained from those contained in the original programs), and Hg-derivability guided by these annotations must be preserved. The following definition formalizes these requirements.

Definition 3 (SA translation): Let $C\in\mathbf{AComm}$, $\phi,\psi,\epsilon\in\mathbf{Assert}$, and $\mathfrak{T}:\mathbf{Assert}\times\mathbf{AComm}\times\mathbf{Assert}\times\mathbf{Assert}\hookrightarrow\mathbf{Assert}\times\mathbf{Comm}^{\text{SA}}\times\mathbf{Assert}\times\mathbf{Assert}$. The function $\mathfrak{T}$ is said to be an *SA translation* if when $\mathfrak{T}(\phi,C,\psi,\epsilon)=(\phi',C',\psi',\epsilon')$, we have that $\phi'\#C'$, and the following hold:

1) If $\models\{\phi'\}\,C'\,\{\psi',\epsilon'\}$, then $\models\{\phi\}\,\lfloor C\rfloor\,\{\psi,\epsilon\}$.
2) If $\vdash_{\mathsf{Hg}}\{\phi\}\,C\,\{\psi,\epsilon\}$, then $\vdash_{\mathsf{Hg}}\{\phi'\}\,C'\,\{\psi',\epsilon'\}$.

In [25] we define a concrete SA translation, and prove that it complies with the above definition. The translation creates a non-iterating program that checks the initialization and preservation of all loops. It takes advantage of the SA features to achieve isolation between parts of the program that encode different axiomatic aspects of the initial program.

As an example of application of a concrete translation, recall the Fact program of Section I, and let $\mathsf{Fact}^{\mathsf{SA}}$ be the program shown in Figure 1d. The latter is the result of translating Fact using a translation conforming to Definition 3. Note that in $\mathsf{Fact}^{\mathsf{SA}}$ the variables f_1, i_1 correspond to the values of f and i after their initialization in the original program; at this point the invariant must be checked ($\mathbf{assert}\,f_1=(i_1-1)!\wedge i_1\leq n_0+1$) to ensure that it has been properly initialized. Next we consider the execution of an arbitrary execution of the loop. The invariant is assumed with the fresh variables f_2, i_2, which ensures isolation from the previous commands. The value of the variables at the end of the iteration is captured by f_3, i_3, and the command $\mathbf{assert}\,f_3=(i_3-1)!\wedge i_3\leq n_0+1$ checks that the invariant is preserved by this arbitrary iteration, which is encoded as the then-branch of a conditional that includes the loop condition in the context. The else-branch ensures that the assumed invariant holds for the loop exit variables f_3, i_3 when the loop terminates.

Finally, a possible SA translation for $\{n\geq 0\}\,\mathsf{Fact}\,\{f=n!,\bot\}$ is now the SA triple $\{n_0\geq 0\}\,\mathsf{Fact}^{\mathsf{SA}}\,\{f_3=n_0!,\bot\}$.

The second component of the deductive workflow is a Verification Conditions Generator (VCGen). This is a function that takes as input an SA Hoare triple, and outputs a set Γ of formulas, such that the formulas in Γ are all valid if and only if the triple can be obtained with system Hsa. The following definition states this in precise terms, and concrete algorithms are presented in [23], [25].

Definition 4 (VCGen for SA triples): A *VCGen for SA triples* is a function $\mathsf{VCG}:\mathbf{Assert}\times\mathbf{Comm}^{\text{SA}}\times\mathbf{Assert}\times\mathbf{Assert}\to\mathcal{P}(\mathbf{Assert})$ such that, when $\Gamma=\mathsf{VCG}(\phi,C,\psi,\epsilon)$ with $\{\phi\}\,C\,\{\psi,\epsilon\}$ an SA Hoare triple, $\models\Gamma$ iff $\vdash_{\mathsf{Hsa}}\{\phi\}\,C\,\{\phi\wedge\psi',\phi\wedge\epsilon'\}$ for some ψ', $\epsilon'\in\mathbf{Assert}$ such that $\models\phi\wedge\psi'\to\psi$ and $\models\phi\wedge\epsilon'\to\epsilon$.

The workflow, shown in Figure 6 (left) can now be formalized by the following two properties. The soundness result states that if the verification conditions (VCs) generated after translating to SA form some annotated version of a Hoare triple are valid, then so is the original triple.

Theorem 1 (Soundness of DV Workflow): Let $C_0\in\mathbf{Comm}$, $C\in\mathbf{AComm}$, and $\phi,\psi,\epsilon\in\mathbf{Assert}$, such that $\lfloor C\rfloor=C_0$ and $\models\mathsf{VCG}(\mathfrak{T}(\phi,C,\psi,\epsilon))$, with $\mathfrak{T}$ and VCG functions satisfying respectively Definitions 3 and 4. Then $\models\{\phi\}\,C_0\,\{\psi,\epsilon\}$.

Proof. Follows the scheme depicted in Figure 6 (center). □

Completeness, on the other hand, states that for any valid triple, there exists some annotated version of it from which a set of valid VCs is generated, after conversion to SA form.

Theorem 2 (Completeness of DV Workflow): Let $C_0\in\mathbf{Comm}$ and $\phi,\psi,\epsilon\in\mathbf{Assert}$ with $\mathbf{Assert}$ expressive wrt. $\mathbf{Comm}$ and the implicit interpretation structure. If $\models\{\phi\}\,C_0\,\{\psi,\epsilon\}$, then there exists some $C\in\mathbf{AComm}$ such that $\lfloor C\rfloor=C_0$ and $\models\mathsf{VCG}(\mathfrak{T}(\phi,C,\psi,\epsilon))$ for any $\mathfrak{T}$,VCG satisfying Definitions 3 and 4.

Proof. Follows the scheme depicted in Figure 6 (right). □

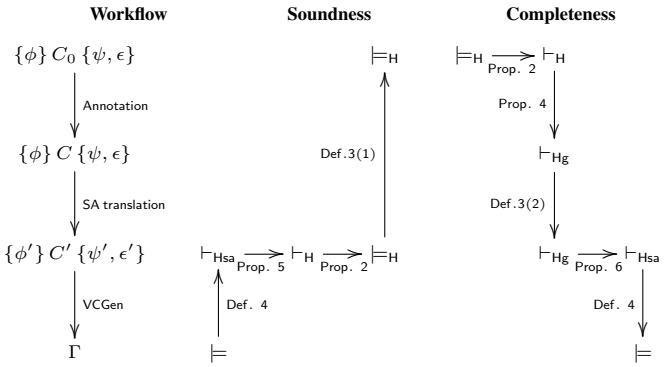


Fig. 6: Soundness and completeness of the DV workflow

V. Bounded Model Checking

Bounded model checking of software basically consists in unwinding loops a certain number of times (k), prior to property checking. In order to ensure the soundness or completeness of this approach, an *unwinding assertion* or *unwinding assumption* must be inserted immediately after the loop unwinding. Unwinding assertions are used to check if there exist executions requiring more than k iterations – if not, this means that the initial program only has bounded executions of length $\leq k$, and the approach is thus sound. If the program has unbounded executions, or if it is not practical to check bounded executions because of their length, the technique can still be applied but it will not be sound. In this case it is necessary for the sake of completeness to add negated loop conditions (known as unwinding assumptions) after loop expansions, which will exclude executions requiring more than k iterations from being considered for verification.

In this section we formalize the two workflows that result from expanding loops including unwinding assertions and unwinding assumptions. We start by defining a translation of iterating programs into non-iterating programs, using a loop unrolling strategy. The translation is carried out by the mutually recursive generic functions $\mathfrak{B}$ and $\mathfrak{U}$ defined below.

Definition 5: The functionals $\mathfrak{B}$ and $\mathfrak{U}$ map every pair $(\alpha, n) \in \{\bullet, \Box\} \times \mathbb{N}$ to functions $\mathfrak{B}^\alpha_n : \mathbf{Comm} \to \mathbf{Comm}$ and $\mathfrak{U}^\alpha_n : \mathbb{N} \times \mathbf{Comm} \hookrightarrow \mathbf{Comm}$ as follows:

$$\mathfrak{B}^\alpha_n(C) = C \ , \ \text{when } C \text{ is } \mathbf{skip}, \mathbf{assume}\,\theta, \mathbf{assert}\,\theta, \text{ or } x := e$$
$$\mathfrak{B}^\alpha_n(C_1\,;\,C_2) = \mathfrak{B}^\alpha_n(C_1)\,;\,\mathfrak{B}^\alpha_n(C_2)$$
$$\mathfrak{B}^\alpha_n(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}) = \mathbf{if}\ b\ \mathbf{then}\ \mathfrak{B}^\alpha_n(C_1)\ \mathbf{else}\ \mathfrak{B}^\alpha_n(C_2)\ \mathbf{fi}$$
$$\mathfrak{B}^\alpha_n(\mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}) = \mathfrak{U}^\alpha_n(n, \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od})$$
$$\mathfrak{U}^\alpha_n(0, \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}) = \begin{cases} \mathbf{assert}\,\neg b & , \text{ if } \alpha = \bullet \\ \mathbf{assume}\,\neg b & , \text{ if } \alpha = \Box \end{cases}$$
$$\mathfrak{U}^\alpha_n(k, \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}) = \mathbf{if}\ b\ \mathbf{then}\ \mathfrak{B}^\alpha_n(C)\,; \mathfrak{U}^\alpha_n(k-1, \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od})\ \mathbf{else\ skip\ fi}$$

In this definition, the first function takes a parameter α that determines whether an unwinding assertion ($\alpha = \bullet$) or unwinding assumption ($\alpha = \Box$) is to be used; a bound n; and a program C. The function is then applied recursively, and in the case of the while construct the function $\mathfrak{U}^\alpha$ is invoked with an additional parameter containing the number of times that the current loop must be unwound (initially fixed at n). Thus, the function $\mathfrak{U}^\alpha$ will unwind the loop and mutually invoke $\mathfrak{B}^\alpha$ to unroll inner loops. An unwinding assertion (resp. assumption) is inserted when the current loop is fully unwound ($n = 0$).

1) Unwinding Assertions Workflow: We will show that the translation using unwinding assertions, $\mathfrak{B}^\bullet_n$, is sound wrt. Hoare logic. In particular, if the result of expanding loops with unwinding assertions is a valid triple, $\models \{\phi\}\, C\, \{\psi, \epsilon\}$ holds whenever $\models \{\phi\}\, \mathfrak{B}^\bullet_n(C)\, \{\psi, \epsilon\}$ for some $n \in \mathbb{N}$.

Before going into this proof some lemmas must be considered. The first one states that it is possible to expand loops further while preserving the final state, when the unwound program does not terminate in the error state.

Lemma 4: Let $n, k_1, k_2, r_1 \in \mathbb{N}$, such that $k_1 \leq k_2 \leq n$.

1) If $\langle \mathfrak{U}^\bullet_n(k_1, C), s\rangle \Rightarrow^{r_1} \sigma$ and $\sigma \neq \bullet$, then $\langle \mathfrak{U}^\bullet_n(k_2, C), s\rangle \Rightarrow^{r_2} \sigma$ for some $r_2 \leq r_1 + 1$.
2) If $\langle \mathfrak{U}^\bullet_n(k_1, C), s\rangle \not\Rightarrow^{r_1}$, then $\langle \mathfrak{U}^\bullet_n(k_2, C), s\rangle \not\Rightarrow^{r_2}$ for some $r_2 \leq r_1 + 1$.

Proof. Both by induction on k_1. □

It is now possible to relate executions of the bounded program and the original program: the original will always terminate in the same state as the bounded program, whenever the latter does not terminate in the error state (Lemma 5). Reversely, if the original terminates in a non-error state, then the bounded program terminates in the same state or in the error state; if the original program terminates in the error state, then the bounded one will also terminate in the error state (Lemma 6). The soundness result can then be proved. In the rest of the paper $\sharp C$ will denote the size (number of constructs) of program C.

Lemma 5: For every $n, r \in \mathbb{N}$ the following both hold:

1) If $\langle \mathfrak{B}^\bullet_n(C), s\rangle \Rightarrow^r \sigma$ and $\sigma \neq \bullet$, then $\langle C, s\rangle \Rightarrow^* \sigma$.
2) If $\langle \mathfrak{B}^\bullet_n(C), s\rangle \not\Rightarrow^r$, then $\langle C, s\rangle \not\Rightarrow^*$.

Proof. Both by induction on $(r, \sharp C)$ using Lemma 4. The proof of 2 also uses 1. □

Lemma 6: For all $n, r \in \mathbb{N}$, the following hold:

1) If $\langle C, s\rangle \Rightarrow^r \sigma$ and $\sigma \neq \bullet$, then $\langle \mathfrak{B}^\bullet_n(C), s\rangle \Rightarrow^* \sigma$ or $\langle \mathfrak{B}^\bullet_n(C), s\rangle \Rightarrow^* \bullet$.
2) If $\langle C, s\rangle \Rightarrow^r \bullet$, then $\langle \mathfrak{B}^\bullet_n(C), s\rangle \Rightarrow^* \bullet$.

Proof. Both by induction on $(r, \sharp C)$. The proof of 2 uses 1. □

Proposition 7 (Soundness of $\mathfrak{B}^\bullet_n$): If $\models \{\phi\}\, \mathfrak{B}^\bullet_n(C)\, \{\psi, \epsilon\}$ for some $n \in \mathbb{N}$, then $\models \{\phi\}\, C\, \{\psi, \epsilon\}$.

Proof. Assume $\models \{\phi\}\, \mathfrak{B}^\bullet_n(C)\, \{\psi, \epsilon\}$, and $s \models \phi$ for some $s \in \Sigma$. $\langle C, s\rangle \Rightarrow^* \bullet$ cannot hold because, by Lemma 6 (2), it contradicts the hypotesis. Therefore, $\models \{\phi\}\, C\, \{\psi, \epsilon\}$ follows from the hypotesis, using Lemma 5. □

Completeness of $\mathfrak{B}^\bullet_n$ does not hold: the validity of $\{\phi\}\, C\, \{\psi, \epsilon\}$ does not imply that $\{\phi\}\, \mathfrak{B}^\bullet_n(C)\, \{\psi, \epsilon\}$ is valid for some $n \in \mathbb{N}$, because executions in states satisfying ϕ may not terminate, and even if all executions terminate, there may exist infinitely many states satisfying ϕ, in which case it may

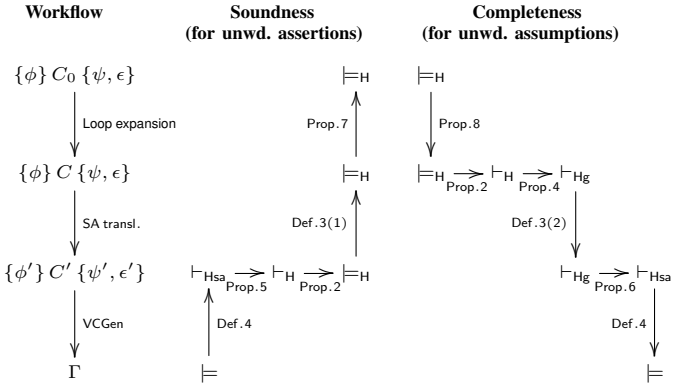


Fig. 7: Soundness and completeness of the BMC workflow

not be possible to find a sufficient value n to mimic the behavior of C in all executions – as an example, consider the valid triple $\{x > 0\}$ **while** $x > 0$ **do** $x := x - 1$ **od** $\{x = 0, \bot\}$.

The soundness of the BMC workflow, which is shown in Figure 7 (left), means that the validity of the VCs generated after translating to SA form the result of expanding the loops in a program, implies that the program is correct.

Theorem 3 (Soundness of BMC Workflow): Let $C_0 \in \mathbf{Comm}$, $\phi, \psi, \epsilon \in \mathbf{Assert}$, and $n \in \mathbb{N}$ such that $\models \mathsf{VCG}(\mathfrak{T}(\phi, \mathfrak{B}^{\bullet}_n(C_0), \psi, \epsilon))$, with $\mathfrak{T}$ and VCG functions satisfying Definitions 3 and 4. Then $\models \{\phi\}\, C_0\, \{\psi, \epsilon\}$.

Proof. Follows the scheme depicted in Figure 7 (center). □

In fact the above result could be formulated in a stronger way since execution of C_0 in states satisfying ϕ is guaranteed to terminate. This corresponds to the notion of *total correctness*, which is not captured by the triple $\{\phi\}\, C_0\, \{\psi, \epsilon\}$.

2) Unwinding Assumptions Workflow: We will first show that the translation $\mathfrak{B}^{\Box}_n$ is complete in the sense that, for every $n \in \mathbb{N}$, $\models \{\phi\}\, \mathfrak{B}^{\Box}_n(C)\, \{\psi, \epsilon\}$ holds whenever $\models \{\phi\}\, C\, \{\psi, \epsilon\}$. In order to prove completeness we need to consider the following lemmas, which state that when a bounded program terminates (i.e. the execution does not diverge or block) for a certain bound k, then the bound can be increased without changing the final state (Lemma 7), and that the termination of a bounded program implies the termination of the original program from which it was obtained, when executed from the same initial state (Lemma 8).

Lemma 7: Let $n, k_1, k_2, r_1 \in \mathbb{N}$, such that $k_1 \leq k_2 \leq n$. If $\langle \mathfrak{U}^{\Box}_n(k_1, C), s\rangle \Rightarrow^{r_1} \sigma$, then $\langle \mathfrak{U}^{\Box}_n(k_2, C), s\rangle \Rightarrow^{r_2} \sigma$ for some $r_2 \leq r_1 + 1$.

Proof. By induction on k_1. □

Lemma 8: For all $n, r \in \mathbb{N}$, if $\langle \mathfrak{B}^{\Box}_n(C), s\rangle \Rightarrow^{r} \sigma$, then $\langle C, s\rangle \Rightarrow^{*} \sigma$.

Proof. By induction on the pair $(r, \sharp C)$, using Lemma 7. □

Proposition 8 (Completeness of $\mathfrak{B}^{\Box}_n$): If $\models \{\phi\}\, C\, \{\psi, \epsilon\}$, then for all $n \in \mathbb{N}$ $\models \{\phi\}\, \mathfrak{B}^{\Box}_n(C)\, \{\psi, \epsilon\}$.

Proof. Assume $\models \{\phi\}\, C\, \{\psi, \epsilon\}$. Let $s \in \Sigma$ such that $s \models \phi$. It is clear that the execution of $\mathfrak{B}^{\Box}_n(C)$ does not diverge since it does not contain loops. If it blocks then we are done, otherwise $\langle \mathfrak{B}^{\Box}_n(C), s\rangle \Rightarrow^{*} \sigma$ for some σ. Clearly $\sigma \neq \bullet$, or else from Lemma 8 we would have that $\langle C, s\rangle \Rightarrow^{*} \bullet$ which contradicts our assumption. If $\sigma = \mathbf{n}(s')$ (resp. $\sigma = \mathbf{e}(s')$) for some $s' \in \Sigma$, then from Lemma 8 $\langle C, s\rangle \Rightarrow^{*} \mathbf{n}(s')$ (resp. $\langle C, s\rangle \Rightarrow^{*} \mathbf{e}(s')$) and thus $s' \models \psi$ (resp. $s' \models \epsilon$). □

This workflow is complete: the formulas obtained by unrolling loops, translating to SA form, and generating VCs from this form, are necessarily valid.

Theorem 4 (Completeness of BMC Workflow): Let $C_0 \in \mathbf{Comm}$, $n \in \mathbb{N}$, and $\phi, \psi, \epsilon \in \mathbf{Assert}$ with **Assert** expressive wrt. **Comm** and the implicit interpretation structure. If $\models \{\phi\}\, C_0\, \{\psi, \epsilon\}$, then $\models \mathsf{VCG}(\mathfrak{T}(\phi, \mathfrak{B}^{\Box}_n(C_0), \psi, \epsilon))$ for any $\mathfrak{T}$, VCG satisfying Definitions 3 and 4.

Proof. Follows the scheme depicted in Figure 7 (right). □

A soundness result (of theoretical interest) can also be stated for this translation. If every bounded expansion of a program C is correct, then C is surely correct. In other words, if C violates its specification, then there exists a bound n such that $\mathfrak{B}^{\Box}_n(C)$ violates that same specification. Lemmas 9 and 10 below are used in the proof of this soundness property.

Lemma 9: Let $n, k_1, k_2, n_1, n_2, r \in \mathbb{N}$, such that $k_1 \leq k_2$, $n_1 \leq n_2$, $k_1 \leq n_1$, and $k_2 \leq n_2$. The following hold:

1) If $\langle \mathfrak{U}^{\Box}_{n_1}(k_1, C), s\rangle \Rightarrow^{r} \sigma$, then $\langle \mathfrak{U}^{\Box}_{n_2}(k_2, C), s\rangle \Rightarrow^{*} \sigma$.
2) If $\langle \mathfrak{B}^{\Box}_{n_1}(C), s\rangle \Rightarrow^{r} \sigma$, then $\langle \mathfrak{B}^{\Box}_{n_2}(C), s\rangle \Rightarrow^{*} \sigma$.

Proof. By simultaneous induction on the pair $(r, \sharp C)$. □

Lemma 10: Let $r \in \mathbb{N}$. If $\langle C, s\rangle \Rightarrow^{r} \sigma$, then $\langle \mathfrak{B}^{\Box}_n(C), s\rangle \Rightarrow^{*} \sigma$ for some $n \in \mathbb{N}$.

Proof. By induction on the pair $(r, \sharp C)$, using Lemma 9. □

Proposition 9 (Soundness of $\mathfrak{B}^{\Box}_n$): If $\models \{\phi\}\, \mathfrak{B}^{\Box}_n(C)\, \{\psi, \epsilon\}$ for every $n \in \mathbb{N}$, then $\models \{\phi\}\, C\, \{\psi, \epsilon\}$.

Proof. Assume $\models \{\phi\}\, \mathfrak{B}^{\Box}_n(C)\, \{\psi, \epsilon\}$. Let $s \in \Sigma$ such that $s \models \phi$. If $\langle C, s\rangle \not\Rightarrow^{*}$ or $\langle C, s\rangle$ diverges, we are done. Otherwise it must be the case that $\langle C, s\rangle \Rightarrow^{*} \sigma$, and from Lemma 10 we have $\sigma \neq \bullet$; if $\sigma = \mathbf{n}(s')$ for some $s' \in \Sigma$ then $s' \models \psi$; if $\sigma = \mathbf{e}(s')$ for some $s' \in \Sigma$ then $s' \models \epsilon$. □

VI. Related Work

1) Deductive Verification: A major trend in deductive verification is to use general-purpose tools that provide VC generation for their specific programming languages. The foremost examples of such tools are Boogie [2] and Why3 [12], whose languages are called BoogiePL and WhyML. Many verification tools rely on one of these, for instance: Spec# [3] (for C# programs) and Havoc [5] (for C programs) use Boogie, and GNATprove [17] (for SPARK/Ada) relies on Why3. Dafny [21] is a language and program verifier toolset that also employs Boogie as VCGen. Both tools use internally the two transformations of programs that we study in this paper. The programming language used here is significantly simpler than BoogiePL or WhyML, but still our deductive workflow can be seen as a model of the Boogie and Why3 workflows.

In deductive verification tools, loops annotated with invariants are commonly encoded as non-iterating programs using *assume* and *assert*. This technique has been used at least since the development of the ESC tools [10], [22], [13], where this was called a "conservative desugaring" (by opposition to

a bounded encoding). In Boogie this has been generalized to non-structured forms of iteration (not considered in our model), using the *goto* command of the BoogiePL language. Back edges are eliminated prior to VC generation by introducing further *assume*/*assert* statements, resulting in acyclic control-flow that can be readily passified (i.e. transformed into SA form). Developers of verifiers for high-level languages can translate loops as BoogiePL loops (desugared into *havoc*/*assume*/*assert* in the internal intermediate forms), or else use their own encoding (the BoogiePL *havoc* command is crucial for this purpose, since the language is not SA).

Soundness results have been given for translations of specific languages into BoogiePL: Lehner and Müller [20] give a translation of a subset of Java bytecode (including loop elimination using *havoc*/*assume*/*assert*), and prove that the weakest precondition of the translated program is stronger than that of the original program (completeness is not discussed). In a paper focusing on the encoding of multiple methods and their specifications, where all proofs are machine-checked, Vogels, Jacobs, and Piessens [28] define a translation of a toy object-oriented language into a BoogiePL-like intermediate language, and prove soundness directly at the level of operational semantics: if some execution of a method is not in accordance with its specification, then at least one execution of the translated program will fail. This approach, motivated by the fact that in BoogiePL "the VC generation is considered an axiomatic definition of the semantics of the language", is close to what we do for our intermediate language, with the differences that: their language is not SA; the VCGen is defined directly, whereas we introduce a Hoare logic for the SA language; and we additionally study completeness.

Passive form has been famously proposed by Flanagan and Saxe [14] as a way to generate VCs of worst-case quadratic size, even when the program has an exponential number of execution paths (as long as exceptions are not used). Passive programs are (dynamic) SA programs where assignments are replaced by *assume* commands; the statement $x := x + 1$ would be written as $\mathbf{assume}\, x_2 = x_1 + 1$. The soundness of the translation into passive form is proved for iteration-free programs in [14] by showing that it preserves the weakest precondition. We remark that Flanagan and Saxe's work, as well as the theoretical foundations of Boogie, are based on a guarded commands language, whose semantics are given by a defined predicate transformer – in other words, the verification conditions *are* the interpretation of a program. In our work in this paper there is a clear distinction between operational and axiomatic semantics, which allows us to consider separately the soundness of program annotation, loop encoding, translation into SA form, and VC generation, as well as appropriate notions of completeness for each of these.

Boogie uses passive form explicitly as an intermediate form; moreover, although loop elimination and conversion to passive form are separate steps, the tool takes advantage of passive form to eliminate *havoc* commands, which become unnecessary. In our approach, elimination of annotated loops and conversion into SA form are performed in a single step, dispensing entirely the use of a *havoc* command. In Why3 programs are not explicitly translated into SA, but unique symbols are implicitly created and merged during the VC generation, in a similar way to SA variables.

In previous work [24] we have proposed an iterating single-assignment language with annotated loops (and without exceptions), in which the strict dynamic SA constraints can be relaxed in a controlled way, by executing special 'update' code after loop iterations. A logic and VCGen (generating compact VCs) was given for this language, as well as a translation of annotated programs into iterating SA programs. We remark that, unlike the present paper, our previous work did not capture the internal functioning of existing tools, which employ a standard SA form where iteration is not allowed.

The VC generation algorithms used by deductive tools are mostly based on weakest precondition calculations. Our work in this paper is independent of the choice of a particular VCGen. Different tools use different algorithms and in a previous work we have investigated different VCGens [23] and proved their equivalence [25].

2) Bounded Model Checking of Software: Bounded verification techniques for software, of which it has been said that they are "a good example of the wonderful liberation we get by dropping the shackles of soundness" [10], have originated in the model checking community [4]. Prominent tools include CBMC [6], LLBMC [26], and ESBMC [15].

Static single-assignment (SSA) form, in which variables cannot occur syntactically more than once as L-values, is commonly used in bounded model checking of software. Programs are translated into SSA form after iteration has been eliminated and function calls have been inlined in a bounded way. A further step is then performed to transform programs into *conditional normal form*, where programs are single-assignment sequences of atomic commands guarded by path conditions. VC generation is immediate from this form, since the guarded atomic command `if (b) C` can be read logically as an implication. Interestingly, this method for generating VCs avoids exponential explosion in a completely different way from the passive form of Flanagan and Saxe.

To the best of our knowledge, bounded verification techniques for software have not been studied from a program semantics perspective, and no semantics-based formalizations of this workflow can be found in the BMC literature. We have established before [23], [25] that the generation of VCs based on the use of conditional normal form is equivalent to the use of predicate transformers. In the present paper we prove soundness and completeness of the remaining parts of the workflow, using a dynamic, rather than static, notion of SA form. This is more efficient regarding the use of variables, but prevents an optimization used by CBMC in VC generation (see [23] for details). Also, our work has the limitation that only structured iteration (combined with try-catch exceptions) is considered, whereas tools like CBMC are able to handle programs containing arbitrary jumps.

VII. Final Remarks

Our results in this paper show that the properties of Hoare logic extend to a verification workflow that includes either a conservative loop encoding based on invariants or bounded loop expansion (in which case either soundness or completeness will be absent), and translation into SA form. We believe these are important contributions with impact on the underlying theory and practical use of tools like Boogie, Why3, and CBMC, based on variations of this technique.

The formalization given here extends to a setting where programs consist of sets of mutually recursive procedures (or alternatively, to classes with a given set of methods). In this setting, in the annotation step of the deductive workflow each procedure/method will be assigned a *contract* (essentially an intended precondition and postcondition). Like loop invariant annotations, contracts play no role in the semantics, or in system H derivations. System H is extended to handle mutual recursion with a rule that will essentially reason about the procedures' bodies by assuming in the context a specification for each procedure, introduced during the construction of derivations in the same way as loop invariants. In system Hg (as well as in Hsa and in VC generation), derivations will make use of the annotated contracts: deriving a triple $\{\phi\}\,\mathbf{call}\ \mathbf{p}\,\{\psi,\epsilon\}$ will require side conditions relating the formulas ϕ, ψ, ϵ with the contract given for procedure $\mathbf{p}$. The use of auxiliary variables can be dispensed with in contracts – a special operator is used in the postcondition to refer to the initial value of variables, which will be mapped to the appropriate variables in the SA intermediate form. This is a good thing, since the use of auxiliary variables is known to break completness in the presence of recursive procedures [19]. In the BMC workflow, recursive procedure calls are treated by inlining the procedure's body a fixed number of times, similarly to loop unwinding.

Acknowledgments: This work was partially supported by the Norte Portugal Regional Operational Programme (NORTE 2020) under the Portugal 2020 Partnership Agreement, through the European Regional Development Fund (ERDF) and also by national funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within project NORTE-01-0145-FEDER-028550 (REASSURE). The first author is supported by the French National Research Organization (project VOCAL ANR-15-CE25-008).

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, 2018.

[2] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[3] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[5] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2007.

[6] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[7] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[9] Stijn de Gouw and Jurriaan Rot. Effectively eliminating auxiliaries. In *Theory and Practice of Formal Methods*, volume 9660 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2016.

[10] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research report 159, Compaq Systems Research Center, 1998.

[11] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[12] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[13] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.

[14] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205. ACM, 2001.

[15] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: an industrial-strength C model checker. In *ASE*, pages 888–891. ACM, 2018.

[16] Paolo Herms, Claude Marché, and Benjamin Monate. A certified multi-prover verification condition generator. In *VSTTE*, volume 7152 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2012.

[17] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. SPARK 2014 and GNATprove - A competition report from builders of an industrial-strength verifying compiler. *STTT*, 17(6):695–707, 2015.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[19] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Asp. Comput.*, 11(5):541–566, 1999.

[20] Hermann Lehner and Peter Müller. Formal translation of bytecode into BoogiePL. *Electr. Notes Theor. Comput. Sci.*, 190(1):35–50, 2007.

[21] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[22] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *ECOOP Workshops*, volume 1743 of *Lecture Notes in Computer Science*, pages 110–111. Springer, 1999.

[23] Cláudio Belo Lourenço, Maria João Frade, Shin Nakajima, and Jorge Sousa Pinto. A generalized approach to verification condition generation. In *COMPSAC (1)*, pages 194–203. IEEE Computer Society, 2018.

[24] Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto. Formalizing single-assignment program verification: An adaptation-complete approach. In *ESOP*, volume 9632 of *Lecture Notes in Computer Science*, pages 41–67. Springer, 2016.

[25] Cláudio Belo Lourenço. *Single-assignment programs verification*. PhD thesis, University of Minho, 2018. Available at http://hdl.handle.net/1822/56332.

[26] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *VSTTE*, volume 7152 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2012.

[27] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, and Francky Catthoor. A practical dynamic single assignment transformation. *ACM Trans. Design Autom. Electr. Syst.*, 12(4):40, 2007.

[28] Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine checked soundness proof for an intermediate verification language. In *SOFSEM*, volume 5404 of *Lecture Notes in Computer Science*, pages 570–581. Springer, 2009.